

PPPPPPPPPPPP		AAAAAAAAAA	TTTTTTTTTTTTTTTT	CCCCCCCCCCCC	HHH	HHH
PPPPPPPPPPPP		AAAAAAAAAA	TTTTTTTTTTTTTTTT	CCCCCCCCCCCC	HHH	HHH
PPPPPPPPPPPP		AAAAAAAAAA	TTTTTTTTTTTTTTTT	CCCCCCCCCCCC	HHH	HHH
PPP	PPP	AAA	TTT	CCC	HHH	HHH
PPP	PPP	AAA	TTT	CCC	HHH	HHH
PPP	PPP	AAA	TTT	CCC	HHH	HHH
PPP	PPP	AAA	TTT	CCC	HHH	HHH
PPP	PPP	AAA	TTT	CCC	HHH	HHH
PPP	PPP	AAA	TTT	CCC	HHH	HHH
PPPPPPPPPPPP		AAA	TTT	CCC	HHH	HHH
PPPPPPPPPPPP		AAA	TTT	CCC	HHHHHHHHHHHHHHHH	HHH
PPPPPPPPPPPP		AAA	TTT	CCC	HHHHHHHHHHHHHHHH	HHH
PPP		AAAAAAAAAAAAAAAA	TTT	CCC	HHHHHHHHHHHHHHHH	HHH
PPP		AAAAAAAAAAAAAAAA	TTT	CCC	HHH	HHH
PPP		AAAAAAAAAAAAAAAA	TTT	CCC	HHH	HHH
PPP		AAA	TTT	CCC	HHH	HHH
PPP		AAA	TTT	CCC	HHH	HHH
PPP		AAA	TTT	CCC	HHH	HHH
PPP		AAA	TTT	CCC	HHH	HHH
PPP		AAA	TTT	CCCCCCCCCCCC	HHH	HHH
PPP		AAA	TTT	CCCCCCCCCCCC	HHH	HHH
PPP		AAA	TTT	CCCCCCCCCCCC	HHH	HHH

```

PPPPPPPP      AAAAAA      TTTTTTTTTT      RRRRRRRR      TTTTTTTTTT      SSSSSSSS
PPPPPPPP      AAAAAA      TTTTTTTTTT      RRRRRRRR      TTTTTTTTTT      SSSSSSSS
PP           PP      AA           AA      TT           RR           RR           TT           SS
PP           PP      AA           AA      TT           RR           RR           TT           SS
PP           PP      AA           AA      TT           RR           RR           TT           SS
PP           PP      AA           AA      TT           RR           RR           TT           SS
PPPPPPPP      AA           AA      TT           RRRRRRRR      TT           SSSSSS
PPPPPPPP      AA           AA      TT           RRRRRRRR      TT           SSSSSS
PP           AAAAAAAAAA      TT           RR           RR           TT           SS
PP           AAAAAAAAAA      TT           RR           RR           TT           SS
PP           AA           AA      TT           RR           RR           TT           SS
PP           AA           AA      TT           RR           RR           TT           SS
PP           AA           AA      TT           RR           RR           TT           SSSSSSSS
PP           AA           AA      TT           RR           RR           TT           SSSSSSSS

```

```

RRRRRRRR      EEEEEEEEEEE      QQQQQQ
RRRRRRRR      EEEEEEEEEEE      QQQQQQ
RR           RR EE           QQ           QQ
RR           RR EE           QQ           QQ
RR           RR EE           QQ           QQ
RR           RR EE           QQ           QQ
RRRRRRRR      EEEEEEEEEEE      QQ           QQ
RRRRRRRR      EEEEEEEEEEE      QQ           QQ
RR   RR       EE           QQ           QQ
RR   RR       EE           QQ           QQ
RR           RR EE           QQ           QQ
RR           RR EE           QQ           QQ
RR           RR EE           QQ           QQ
RR           RR EEEEEEEEEEE      QQQQ      QQ
RR           RR EEEEEEEEEEE      QQQQ      QQ

```

PATRTS.REQ - RunTime Symbol Table Literals and Structures
Version V03-002

COPYRIGHT (c) 1978, 1980, 1982, 1984 BY
DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.
ALL RIGHTS RESERVED.

THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
TRANSFERRED.

THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
CORPORATION.

DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.

Revision History:

NO --	DATE ----	PROGRAMMER -----	PURPOSE -----
00	20-DEC-77	K.D. MORSE	ADAPT VERSION V04-000 FOR PATCH.
01	27-DEC-77	K.D. MORSE	CHANGE DBG\$ RST BEGIN TO BE GLOBAL PAT\$GL RST BEGN, THAT IS INITIALIZED IN PAT\$FREEINIT.
02	11-JAN-78	K.D. MORSE	ADD FIELDS FOR DSTR ACCESS.
03	28-FEB-78	K.D. MORSE	ADD RST SIZE LITERALS.
04	08-MAR-78	K.D. MORSE	ADD SYMBOL DSC\$W MAXLEN DUE TO NEW SRMDEF.MDL WHICH DOES NOT INCLUDE IT.
05	18-MAR-78	K.D. MORSE	REMOVED ALL DSC DEFINITIONS AS THEY ARE NOW IN SRMDEF.MDL EXCEPT PATCH MIN AND MAX VALUES.
06	06-apr-78	K.D. MORSE	CHANGE GST E NAME ADDR FROM 10 TO 12. ADD NEW FIELDS FOR MC GBLxxx AND NEW DSC\$... FOR GSD TYPES. NEW MC GBL LOCKED FIELD TELLS SYM TO VAL WHEN IT IS OK TO CONSULT THE GST.

Modified by:

V03-002 MTR0022 Mike Rhodes 3-May-1983
Change runtime symbol table name table entry size to correctly
reflect symbol size (symbols upto 31 characters).

V03-001 MTR0012 Mike Rhodes 16-Aug-1982
Modify file names to remove duplicate file names between code
and require files.

1.07 CNH0027 Chris Hume 6-Jan-1980 18:00
Pulled some definitions for the DSC structure that were added
to the system libraries.

--

```
!++
Since the PATCH free-storage manager currently
works in 'units', we define the following macro to
convert a byte-unit quantity into whatever units it
requires. We expect to change the free-storage manager
to work in byte units, so eventually this macro should
just reduce to its actual parameter. For now, however,
it 'rounds up' to the smallest number of LONGWORDS
which are required to contain the indicated number of
bytes.
--
```

```
MACRO      RST_UNITS( bytes ) =
           ( ((bytes) + %upval-1)/%upval )
           %;
```

```
! MACROS:
```

```
MACRO      YES_NO( question )
           ! Ask a question and return the Y/N answer.
           =
           QUERY( UPLIT( %ASCIC question )) %;
```

++
RST-Pointers

So-called RST-pointers are referred to throughout the RST code. They are simply the means of access to RST data structures, and we purposely talk of them as if they were their own TYPE so that we can change this implementation detail if/when we feel it is necessary.

For now, RST-pointers are 16-bit items which are manipulated by the special RST storage routines PAT\$RST_FREEZ and PAT\$RST_RELEASE. No code outside of the RST-DST/PATCH interface module knows anything more about the implementation of RST-pointers than that. (Other modules declare and use RST-pointers via macros, etc.)

If any change is to be made to what RST-pointers actually are, there are only 2 criterion that the new ones much uphold:
1) RST-pointers must be storable in the NT, MC, SAT and LVT fields which are defined for them, and 2) they must be able to provide access to the RST_NT and RST_MC structures defined below.

The following macro is provided so that one can declare REFs to such pointers. Some code also applies %SIZE to this macro to get the size of an RST-pointer. Note that no code should declare an occurrence of an RST-pointer, since we do not define that you can do anything meaningful with such a thing. This is because we want to enforce the usage of REFs to the structures we declare to access RST data structures. (e.g. we use "REF MC_RECORD" to say that we are declaring a pointer to an MC record. REFs to MC_RECORDS also happen to be RST-pointers, but we don't want to build-in this coincidental characteristic.)

--
MACRO

RST_POINTER = VECTOR[1,WORD] %;

++
Pathnames

Symbols in PATCH are actually made up of sequences of symbols or "elements". The concatenation of such elements, along with the element separation character (\), make up a so-called pathname because the sequence represents the path which one must make thru RST data structures to get to the desired symbol.

We represent strings internal to PATCH by passing around so-called counted string pointers. They are simply LONGWORD pointers to a count byte followed by that many characters. The CS_POINTER macro allows us to declare occurrences, REFs, and take the %SIZE of this type of datum.

Pathnames, then, are represented with vectors of CS_POINTERS. Like duplication vectors, they terminate with a 0 entry for programming ease, but also have a maximum size so that we can declare them LOCALLY.

The following macros are used in declarations to not build-in the above conventions.

MACRO

```
CS_POINTER      = REF VECTOR[1,BYTE] %;
```

! PATCH tells the RST module about ASCII
! strings by passing a counted string pointer.

! Symbol pathnames are 0-ended vectors
! of CS_POINTERS. There is a maximum
! length to pathnames so that routines can
! declare LOCAL vectors of pathname pointers.

LITERAL

```
MAX_PATH_SIZE  = 10;
```

MACRO

```
PATHNAME_VECTOR = VECTOR[ MAX_PATH_SIZE +1, %SIZE(CS_POINTER) ] %;
```

++
Overall Characteristics of the RST/DST, etc.
--

++
The PATCH Runtime Symbol Table (RST) free-storage area begins at a fixed virtual address. This GLOBAL is used directly by some of the RST structures since RST-pointers need this information.
--

LITERAL

! THE RST IS A FIXED SIZE BUT THIS FACT IS ONLY USED TO
! ALLOW THE INITIALIZATION OF THE OTHER SIZE LITERALS.
! THESE LITERALS ARE USED TO LIMIT THE AMOUNT OF RST STORAGE
! ALLOCATED FOR VARIOUS PARTS OF THE RST.

RST_TOTAL_SIZE = 65000, ! RST IS 65K BYTES

! THE "SET MODULE" COMMAND DOES NOT ALLOCATE ALL THE FREE STORAGE
! AVAILABLE IN THE RST. INSTEAD, MODULES WILL BE ADDED AS LONG AS
! THERE IS AT LEAST RST_AVAIL_SIZE BYTES AVAILABLE IN THE RST.

RST_AVAIL_SIZE = 3000, ! STORAGE LEFT FOR PATCH ITSELF

! DURING THE RST INITIALIZATION, SPACE IS TAKEN TO SET AS MANY MC'S
! AS WILL LEAVE RST_MODU_SIZE BYTES FOR SUBSEQUENT SET MODULE COMMANDS.
! CURRENTLY THE MC SPACE IS 50% OF THE RST.

RST_MODU_SIZE = (RST_TOTAL_SIZE-RST_AVAIL_SIZE)/2,

! The SAT and LVT are allocated contiguous storage
! by tallying up the number of entries in each table
! we will get from each module we are to initialize.
! The following two minimums are used to begin the
! tally so that the tables will actually be somewhat
! larger than what the MC data implies. The SAT and LVT
! minimums must be at least 1 so that we will never ask
! the free storage manager for 0 bytes.

SAT_MINIMUM = 10, ! Minimum number of SAT entries.
LVT_MINIMUM = 10, ! Minimum number of LVT entries.

! The NT, however, has no such fixed size. MC statistics
! gathering tallies up the number of NT entries, though;
! we begin such a tally at NT_MINIMUM.

NT_MINIMUM = 0, ! Minimum number of NT entries.

! We will use byte indices to fetch RST-pointers to the NT
! from the NT hash vector. This vector, then, must contain
! NT_HASH_SIZE entries, each of which must be large enough
! to store an RST-pointer. See BUILD_RST() in DBGRST.B32
! Also see field NT_FORWARD of the NT-record definition,
! and the corresponding warning in the routine UNLINK_NT_RECS.

```
NT_HASH_SIZE    = %X'FF',      ! NT hash vector size.
```

```
! We will never print "symbol+offset" when the  
! upper bound for "symbol" is 0 and when  
! the offset is greater then RST_MAX_OFFSET
```

```
RST_MAX_OFFSET  = %X'100';
```

```
!+  
! Since scope definitions are recursive, we must  
! stack ROUTINE BEGINS in the routine ADD_MODULE.  
! It is no coincidence that this stack limit is the  
! same as the limit on the length (in elements) of  
! symbol pathnames.  
!-
```

```
LITERAL  
MAX_SCOPE_DEPTH = MAX_PATH_SIZE;      ! Routines can be nested to a maximum depth.
```

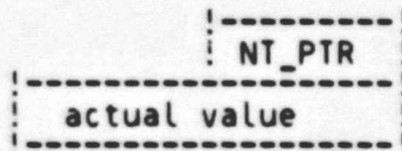
Descriptors

Just as the SRM defines various 'system wide' descriptor formats, the RST modules use a few more descriptors of its own invention. They are as follows:

Value Descriptors

Value Descriptors are used to pass around all needed information about a value which has been obtained from the RST data base. For now they are simply 2-longword blocks:

!-----longword-----!



Value Descriptors must be accessed via the following field names.

```

FIELD
  VALU_FIELD_SET =
SET
  VALU_NT_PTR    = [ 0,0,16,0 ],      ! Associated NT pointer.
  VALU_VALUE     = [ 2,0,32,0 ]      ! The actual value.
TES;
```

Declare an occurrence or REF to a VALUE_DESCRIPTOR via the following macros.

```

LITERAL
  VALU_DESC_SIZE = 8;                ! Each one is 2 longwords long.
```

```

MACRO
  VALU_DESCRIPTOR = BLOCK[ VALU_DESC_SIZE, BYTE ] FIELD( VALU_FIELD_SET ) %;
```

++
Array Bounds Descriptor

An array bounds Descriptor is used to pass around all needed information about an array and its associated dimensions. Like VALU_DESCRIPTORs, they are simply 2-longword blocks, but this might change.

!-----longword-----!

address of array
length of array

Such Descriptors must be accessed via the following field names.

FIELD

```
ARRAY_BNDS_SET =  
SET  ARRAY_ADDRESS = [ 0,0,32,0 ],      ! Beginning address of array.  
      ARRAY_LENGTH  = [ 4,0,32,0 ]      ! Size, in bytes, of array.  
TES;
```

+
Declare an occurrence or REF to an array bounds descriptor via the following macros.

LITERAL

```
ARRAY_BNDS_SIZE = 8;          ! Each one is 2 longwords long.
```

MACRO

```
ARRAY_BNDS_DESC = BLOCK[ ARRAY_BNDS_SIZE, BYTE ] FIELD( ARRAY_BNDS_SET ) %;
```

|++

Vector Storage Descriptors

So-called "vector storage" is the storage which we allocate in relatively large chunks for the explicit purpose of subsequently re-allocating the same storage to someone else in smaller, variable-sized chunks.

This facility has been implemented to interface between the way that the standard PATCH storage manager works, with the way that the RST routines really want to 'allocate' storage. We satisfy the former by only asking for large chunks (and paying the associated overhead), and we satisfy the latter by 'doling' out small-sized chunks with little overhead. We can do this because we never have to free up these chunks so don't have to store the would-be-needed pointers, etc.

!--%size(RST_POINTER)--!

!--(i.e. word)-----!

PTR type
beginning of STORage
end of STORage
nxt free rec in STOR

Such descriptors are accessed via the following field names.

The 'begin' field is the one which various routines look at to decide if the field descriptor is valid.

FIELD

```

SET  STOR_DESC_SET =
      STOR_LONG_PTRS = [ 0,0, 8,0 ],      ! Pointer type. 1 => full word pointers,
                                             ! 0 => RST-pointer access.
      STOR_BEGIN_RST  = [ 1,0,16,0 ],      ! RST pointer to beginning of storage.
      STOR_END_RST    = [ 3,0,16,0 ],      ! RST pointer to end of storage.
      STOR_MARKER     = [ 5,0,16,0 ]      ! Current place in storage.
                                             ! (RST pointer to next available byte).

TES;
```

|+

Declare an occurrence or REF to a vector storage descriptor via the following macros.

LITERAL

PATRTS.REQ;1

16-SEP-1984 16:52:48.93⁴ Page 11

STOR_DESC_SIZE = 7;

! 3 RST pointers take 6 bytes;
! the pointer-type byte takes 1 more.

MACRO

VECT_STORE_DESC = BLOCK[STOR_DESC_SIZE, BYTE] FIELD(STOR_DESC_SET) %;

++ The Module Chain (MC) is a chain of fixed-size records each of which has an RST_MC structure:

!<byte><byte>!<byte><byte>!

x!flags!type	Next MC
DST Pointer	
number of NT entries	
first name bytes ! count	
more name bytes	
more name bytes	
more name bytes	
vector storage descriptor for NT recs	
vector storage descriptor for SAT recs	
vector storage descriptor for LVT recs	
number of SAT entries	
number of LVT entries	

The reason for using our own structure here, (instead of a BLOCK), is because we access MC records with RST-pointers.

LITERAL

```
RST_MC_SIZE      = 57;      ! MC records are fixed-size.
                        ! Each one takes this many bytes.
```

STRUCTURE

```
RST_MC [ off, pos, siz, ext; N=1, unit=1 ] =
  [ N * RST_MC_SIZE ]
  BEGIN
  (
    EXTERNAL PAT$GL_RST_BEGN;
    RST_MC + .PAT$GL_RST_BEGN
  ) + off*unit
  END <pos, siz, ext>
  ;
```

++ MC records have the following fields.

FIELD

SET MC_FIELD_SET =

! **** Some fields (up to NAME_ADDR) must be alligned
! with the corresponding ones in RST_NT structures.

MC_NEXT	= [0,0,16,0],	Next MC record in chain.
MC_TYPE	= [2,0, 8,0],	DST record type byte. Must be DSC\$K_DTYPE_MOD
MC_IS_GLOBAL	= [3,0, 1,1],	0, for 'normal' MCs; 1 for the MC record we 'hang' globals off.
MC_IN_RST	= [3,1, 1,1],	Whether or not this module has been initialized into the RST.
MC_IS_MAIN	= [3,2, 1,1],	Whether or not this module contains the program's transfer address.
MC_GBL_LOCKED	= [3,2, 1,1],	1 = Do not look at GST, 0 = OK. This is for SYM_TO_VAL, and applies to the global MC only.
MC_LANGUAGE	= [3,3, 3,0],	3-BIT encoding of the language which the module is written in.
MC_IS_DYING	= [3,6, 1,0],	Vector storage for this MC is about to be freed up.
MC_not_free	= [3,7, 1,0],	Used in NTs only.
MC_DST_START	= [4,0,32,0],	Record ID of first record for this module.
MC_GBL_NT_PTR	= [4,0,16,0],	Pointer to the single 'fake' NT record for globals. NOTE: NT_PTR and DST_START are mutually exclusive.
MC_NAMES	= [8,0,32,1],	Number of NT records required.
MC_NAME_CS	= [12,0, 8,0],	Name of Module is a counted string. A dotted reference to this field picks up the count, an undotted one addresses the counted string.
MC_NAME_ADDR	= [13,0, 8,0],	The name string itself. An undotted reference to this field addresses only the MC name, a dotted reference picks up the 1st character of the name.

! *** leave up to byte 27 inclusive for _NAME_ field.

MC_NT_STORAGE	= [28,0, 8,0],	Vector storage descriptor for NT records. A direct reference to this field is equivalent to the STOR_LONG_PTRS field of the storage descriptor.
---------------	------------------	--

! *** leave up to byte 34 inclusive for _NT_STORAGE field.

MC_SAT_STORAGE	= [35,0, 8,0],	Vector storage descriptor for SAT records. A direct reference to this field is equivalent to the STOR_LONG_PTRS field of the storage descriptor.
----------------	------------------	---

! *** leave up to byte 41 inclusive for _SAT_STORAGE field.

```
MC_LVT_STORAGE = [ 42,0, 8,0 ],      | Vector storage descriptor for LVT records.  
                                       | A direct reference to this field is  
                                       | equivalent to the STOR_LONG_PTRS  
                                       | field of the storage descriptor.
```

```
! *** leave up to byte 48 inclusive for _LVT_STORAGE field.
```

```
MC_STATICS      = [ 49,0,32,1 ],      | Number of SAT records required.  
MC_LITERALS     = [ 53,0,32,1 ],      | Number of LVT records required.  
MC_GBL_SAT_PTR  = [ 53,0,32,0 ]      | Pointer to the single 'fake'  
                                       | SAT record for globals. NOTE:  
                                       | SAT_PTR and LITERALS are  
                                       | mutually exclusive.
```

```
TES;
```

```
!+  
!- You declare an occurrence or REF of an MC datum via:
```

```
MACRO  
    MC_RECORD      = RST_MC[ RST_MC_SIZE, BYTE ] FIELD( MC_FIELD_SET ) %;
```

++ The Name Table (NT) is a set of doubly-linked records with the following format:

!<byte><byte>!<byte><byte>!

x!flags!type	Next NT
DST Pointer	
back hash	forw hash
first name bytes ! count	
more name bytes	
more name bytes	
more name bytes	

Since access to such records will be via so-called RST-pointers, (16-bit pointers which we always add a global to before using), we define the following structure to localize this implementation detail.

++

```
LITERAL
RST_NT_OVERHEAD = 13,      | Number of bytes in NT record excluding those
                             | taken up by the name. (So that this
                             | number + .NT_PTR[ NT_NAMES_CS ] gives
                             | the length of the NT record in bytes.)
                             | (This is solely for the benefit of routines
                             | unlink_nt_recs, add_nt, and add_gst_nt.)
RST_NT_SIZE      = 44;     | A static NT record would take a max # of bytes.
                             | (Dynamically-allocated ones usually take less).
```

```
STRUCTURE
RST_NT [ off, pos, siz, ext; N=1, unit=1 ] =
  [ N * RST_NT_SIZE ]
  BEGIN
  (
  EXTERNAL PAT$GL RST_BEGN;
  RST_NT + .PAT$GL_RST_BEGN
  ) + off*unit
  END <pos, siz, ext>
  ;
```

++ Access to an NT chain is via a 'hash' vector. Conceptually, this is a vector of RST-pointers, and we define the following macro to declare REfs or occurrences of these elements. (because we may decide to change their representation)

!-

MACRO

NT_HASH_RECORD = VECTOR[1,WORD] %;

!+

NT records have the following fields.

Note that NT_FORWARD must be the first field in the record so that unlink_nt_recs can overlay NT_FORWARD and a given entry in the NT_HASH_VECTOR.

FIELD

NT_FIELD_SET =

SET

! **** Some fields (up to NAME_ADDR) must be alligned with the corresponding ones in RST_MC structures.

NT_FORWARD = [0,0,16,0],

NT_TYPE = [2,0, 8,0],

NT_IS_GLOBAL = [3,0, 1,1],

NT_NOT_FREE = [3,1, 6,0],

NT_IS_BOUNDED = [3,7, 1,0],

NT_DST_PTR = [4,0,32,0],

NT_GBL_VALUE = [4,0,32,0],

NT_UP_SCOPE = [8,0,16,0],

NT_BACKWARD = [10,0,16,0],

NT_NAME_CS = [12,0, 8,0],

NT_NAME_ADDR = [13,0, 8,0]

Next NT record in hash chain.
 FORWARD must be first. See above.
 DST record type byte, (from SRM),
 or unused if NT_IS_GLOBAL.
 Whether or not the symbol is GLOBAL.
 Used in MCs but not in NTs.
 Used in NTs only. => symbol's
 LB and UB are not 0.
 Pointer to associated DST record.
 Value of symbol when it is
 bound only to a GST record.
 Pointer to NT record for symbol
 that is 'above' this as far as
 scope is concerned.
 Backward NT hash chain link.
 Name of symbol is a counted string.
 A dotted reference to this field picks
 up the count, an undotted one
 addresses the counted string.
 The name string itself. An undotted
 reference to this field addresses
 only the MC name, a dotted reference
 picks up the 1st character of the name.

TES;

!+

You define an occurrence or REF to an NT record via:

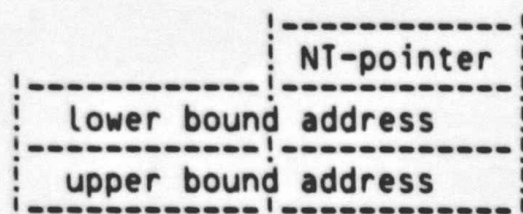
MACRO

NT_RECORD = RST_NT[RST_NT_SIZE, BYTE] FIELD(NT_FIELD_SET) %;

++

The Static Address Table (SAT) is a vector of fixed-size records (blocks) with the following format:

!<byte><byte>!<byte><byte>!



The lower and upper bound address fields contain the beginning and ending virtual addresses which were bound to the symbol by the linker. The NT-pointer field contains an RST-pointer into the name table (NT) for the NT entry which corresponds to this symbol.

Overall Structure:

Logically, the SAT is a sequence of fixed-size records ordered on the `_UB` field so that we can search them sequentially. Physically the storage is actually discontinuous, space being associated with the module the space was allocated on behalf of. Sequentially access to the SAT is that which is provided and defined by `GET_NEXT_SAT` in the following manner:

```

1) call GET_NEXT_SAT( SL_ACCE_INIT )
   to set up to begin scanning the SAT
then
2) call ptr = GET_NEXT_SAT( access_type )
   to have 'ptr' set to the next SAT record, where
   the notion of 'next' is defined by 'access_type'.
```

Currently 3 access types are defined. `_RECS` and `_SORT` both ask for the next sequential record in a logical sense. (i.e. records marked for deletion are quietly skipped over). The ending criterion for `_RECS` access is that there are no more records left, while `_SORT` access, expected to be used with the 'shell' sort, ends each time like `_RECS` does but at that time causes the access routine to restore the context which it saved after the last `_SORT` call so that subsequent `_RECS` calls scan from where they left off last time. In both cases 0 is returned in 'ptr' when there are no more records for the indicated access type.

For the type of sequential access we need when moving endangered SAT/LVT records to storage not DYING, we also define a third access mode called `SL_ACCE_FREE`.

This mode asks for modules IN_RST AND IS_DYING to be skipped over so that only pointers to 'safe' records are returned.

In all cases, the same INIT code must be used to 'start off' the access sequence, and no concurrent accessing is allowed except for the limited type supported via RECS/SORT.

LITERAL

```
SL_ACCE_INIT    = 0,    ! See above.  "SL" --> SAT/LVT
SL_ACCE_RECS    = 1,
SL_ACCE_SORT    = 2,
SL_ACCE_FREE    = 3;
```

+ SAT/LVT Correspondence

While the SAT and LVT are as similar in structure as they are now, the two are manipulated by the same routines as much as possible. This will remain OK as long as the fields which must correspond still do. See the "Implicit Inputs" section of the common routines for details.

+ SAT records have the following fields.

FIELD

```
SAT_FIELD_SET =
SET
| **** The SAT and LVT structures must be aligned so that
| the NT_PTR fields match, and so that the LB and VALUE
| fields overlap. The latter must be true only as long
| as the two share a common sort routine which relies on
| this alignment. The former must be true as long as
| the two share any routines which access SAT_NT_PTR
| (COMPRES_SAT_LVT, DELE_SAT_LVT, etc).
|
SAT_NT_PTR      = [ 0,0,16,0 ],    ! Points to associated NT record.
SAT_LB          = [ 2,0,32,0 ],    ! Lower bound static address.
SAT_UB          = [ 6,0,32,0 ]     ! Upper bound static address.
TES,
```

+ You declare an occurrence or REF of an SAT datum via the macro, SAT_RECORD. If you want the %SIZE of a pointer to such a thing, use %size(SAT_POINTER).

LITERAL

```
RST_SAT_SIZE    = 10;    ! Each SAT record takes this many bytes.
```

MACRO

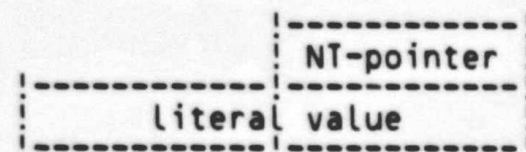
PATRTS.REQ;1

16-SEP-1984 16:52:48.^D₅93 Page 19

SAT_RECORD = BLOCK[RST_SAT_SIZE, BYTE] FIELD(SAT_FIELD_SET) %,
SAT_POINTER = REF BLOCK[RST_SAT_SIZE, BYTE] %;

++
The Literal Value Table (LVT) is a vector of fixed-size LVT records each of which has the following format:

!<byte><byte>!<byte><byte>!



The value field contains the longword value which is bound to the literal.

The NT-pointer is an RST-pointer to the NT record for this symbol.

Overall Structure:

Logically, the LVT is a sequence of fixed-size records ordered on the VALUE field so that we can search them sequentially. Physically the storage is actually discontinuous, space being associated with the module the space was allocated on behalf of. Sequentially access to the LVT is that which is provided and defined by GET_NEXT_LVT using the same control literals and the same mechanisms as are described for the SAT, above.

+
LVT records have the following fields.

FIELD

LVT_FIELD_SET =

SET

! **** The SAT and LVT structures must be alligned so that the NT_PTR fields match, and so that the LB and VALUE fields overlap. The latter must be true only as long as the two share a common sort routine which relies on this alignment. The former must be true as long as the two share any routines which access SAT_NT_PTR (COMPRES_SAT_LVT, DELE_SAT_LVT, etc).

LVT_NT_PTR = [0,0,16,0], ! Pointer to associated NT record.
LVT_VALUE = [2,0,32,0] ! Value bound to the literal.

TES;

+
You declare an occurrence or REF of an LVT datum via:

LITERAL
RST_LVT_SIZE = 6; ! Each LVT record takes this many bytes.

PATRTS.REQ;1

16-SEP-1984 16:52:48.^F₅93 Page 21

MACRO

LVT_RECORD = BLOCK[RST_LVT_SIZE, BYTE] FIELD(LVT_FIELD_SET) %;

```

!++
! The format of DST records is defined as a BLOCK
! (even though the records are variable sized)
! with the following fields:
!--

```

```

FIELD
SET  DSTR_FIELD_SET =
      DSTR_SIZE      = [ 0,0, 8,0 ],      ! First byte is record size in bytes.
      DSTR_TYPE      = [ 1,0, 8,0 ],      ! Type of DST record.
      DSTR_ACCESS    = [ 2,0, 8,1 ],      ! Access field.

      !++
      ! Subfields of ACCESS are offset from the beginning
      ! of the DSTR record.
      !--
      DSTR_ACCES_TYPE = [ 2,0, 2,0 ],      ! Type of access
      DSTR_ACCES_BASED = [ 2,2, 2,0 ],      ! Based or not
      DSTR_ACCES_BREG = [ 2,4, 4,0 ],      ! Associated register

      DSTR_ACC01      = [ 2,0, 2,0 ],      ! Access field bits 0+1.
      DSTR_ACC23      = [ 2,2, 2,0 ],      ! Access field bits 2+3.
      DSTR_ACC47      = [ 2,4, 4,0 ],      ! Access field bits 4-7.
      DSTR_VALUE      = [ 3,0,32,1 ],      ! Value field contains various things,
                                           ! but they are always longwords.
      DSTR_NAME       = [ 7,0, 8,0 ]       ! The symbol name field is a counted string.
                                           ! (The field ref picks up the count.)

```

```

TES;

```

```

!+
! You declare an occurrence or REF of a DST datum via:
!-

```

```

LITERAL  DST_RECORD_SIZE = 23;    ! Each DST record is at most 23 bytes long.

```

```

MACRO    DST_RECORD = BLOCK[ DST_RECORD_SIZE, BYTE] FIELD( DSTR_FIELD_SET ) %;

```

```

!++
! The DST record types are defined in SRM as 'system-wide' names.
! The fields for the DSC structure are:

```

```

dsc$b_length      = 0, 0, 16, 1%,      ! length of string
dsc$b_dtype       = 2, 0, 8, 0%,      ! type of data
dsc$b_class       = 3, 0, 8, 0%,      ! type of descriptor
dsc$a_pointer     = 4, 0, 32, 0%,      ! address of string
dsc$w_maxlen      = 8, 0, 16, 1%,      ! maximum length of string

dsc$b_aflags      = 10, 0, 8, 0%,      ! array flag bits
dsc$v_fl_column   = 10, 5, 1, 0%,      ! FORTRAN array element storage
dsc$v_fl_coeff    = 10, 6, 1, 0%,      ! coefficients present
dsc$v_fl_bounds   = 10, 7, 1, 0%,      ! bounds information present
dsc$b_dimct       = 11, 0, 8, 0%,      ! number of dimensions

```

```

dsc$l_arsize = 12, 0, 32, 0%,      ! total size of array in bytes
dsc$a_a0     = 16, 0, 32, 0%,      ! address of element A (0, 0, ... )
dsc$l_m1     = 20, 0, 32, 0%,      ! addressing coefficients

```

```

LITERAL
dsc$k_class_a = 4;                  ! class = descriptor

```

We divide these types essentially into 3 classes
because when we are interested in what type a record
is we usually want to first divide the range into

- 1) BLISS type ZERO,
- 2) SRM atomic types (1-23),

and

- 3) 'the rest' - those which exist for DEBUG and PATCH only.

This is what the _HIGHEST and _LOWEST symbols are for.

```

!MACRO
DSC$W_MAXLEN = 8, 0, 16, 1%; ! MAXIMUM LENGTH OF STRING FOR DESCRIPTOR

```

LITERAL

! Variable types:

```

! DSC$K_DTYPE_Z = 0,      ! Unspecified.
!
DSC$K_DTYPE_V = 1,      ! Bit.
DSC$K_DTYPE_BU = 2,     ! Byte logical.
DSC$K_DTYPE_WU = 3,     ! Word logical.
DSC$K_DTYPE_LU = 4,     ! Longword logical.
DSC$K_DTYPE_QU = 5,     ! Quadword logical.
DSC$K_DTYPE_B = 6,      ! Byte integer.
DSC$K_DTYPE_W = 7,      ! Word integer.
DSC$K_DTYPE_L = 8,      ! Longword integer.
DSC$K_DTYPE_Q = 9,      ! Quadword integer.
DSC$K_DTYPE_F = 10,     ! Single-precision floating.
DSC$K_DTYPE_D = 11,     ! Double-precision floating.
DSC$K_DTYPE_FC = 12,    ! Complex.
DSC$K_DTYPE_DC = 13,    ! Double-precision Complex.
DSC$K_DTYPE_T = 14,     ! ASCII text string.
DSC$K_DTYPE_NU = 15,    ! Numeric string, unsigned.
DSC$K_DTYPE_NL = 16,    ! Numeric string, left separate sign.
DSC$K_DTYPE_NLO = 17,   ! Numeric string, left overpunched sign.
DSC$K_DTYPE_NR = 18,    ! Numeric string, right separate sign.
DSC$K_DTYPE_NRO = 19,   ! Numeric string, right overpunched sign.
DSC$K_DTYPE_NZ = 20,    ! Numeric string, zoned sign.
DSC$K_DTYPE_P = 21,     ! Packed decimal string.
DSC$K_DTYPE_ZI = 22,    ! Sequence of instructions.
DSC$K_DTYPE_ZEM = 23,   ! Procedure entry mask.

```

```

DST_TYP_HIGHEST = 23,      ! Highest TYPE type we support.
                        ! (for range checking)

```

! DST Record Types:

DST_DST_LOWEST = 183, ! Lowest DST type we support.

```

! DSC$K_DTYPE_FLD = 183, ! BLISS FIELD names.
! DSC$K_DTYPE_PCT = 184, ! PSECT information.
! DSC$K_DTYPE_DPC = 185, ! PC correlation table for FORTRAN IV+
! DSC$K_DTYPE_LBL = 186, ! LITERAL or LABEL, depending on
!                       ! another field in the DST.
! DSC$K_DTYPE_SLB = 187, ! Labels in non-MARS modules.
! DSC$K_DTYPE_MOD = 188, ! Beginning of a new MODULE.
! DSC$K_DTYPE_EOM = 189, ! End of Module.
! DSC$K_DTYPE_RTN = 190, ! Beginning of a new ROUTINE.
! DSC$K_DTYPE_EOR = 191, ! End of ROUTINE marker.

```

DST_DST_HIGHEST = 191; ! Highest DST type. (for range checking)
! For now we have our own made-up types to match the GSD
! types 'entry' and 'symbol definition'.

```

! DSC$K_DTYPE_ENT = 192, ! ENTRY (routine) global definition
! DSC$K_DTYPE_GBL = 193; ! Global symbol definition.

```

!++
! The DSTR_ACC01 field from DST_RECORDs has the following possible values:

```

LITERAL
ACCS_LITERAL      = 0,      ! Literal value
ACCS_VALUEADR     = 1,      ! Address of a value
ACCS_DESCRIPTOR   = 2,      ! Address of a symbol's descriptor
ACCS_REGISTER     = 3;      ! Register number

```

!++
! The DSTR_ACC23 field from DST_RECORDs has the following possible values:

```

LITERAL
ACCS_NOTBASDIR    = 0,      ! Not-based, direct
ACCS_NOTBASIND    = 1,      ! Not-based, indirect
ACCS_BASDIR       = 2,      ! Based, direct
ACCS_BASIND       = 3;      ! Based, indirect

```

!++
! The following literals define various fixed-value fields found within DST records.

```

LITERAL
! Each MODULE record indicates the source code language:
MARS_MODULE       = 0,      ! Module is written in MARS.
BLISS_MODULE      = 2,      ! Module is written in BLISS.
FORTRAN_MODULE    = 1;      ! Module is written in FORTRAN.

```

```

!++
! The format of GST records is defined as a BLOCK
! (even though the records are variable sized)
! with the following fields:
!--

```

```

FIELD
SET  GST_FIELD_SET =
      GST_ENTRY_TYPE = [ 0,0, 8,0 ],      ! Type of GST record.
      GST_IS_DEFN    = [ 2,1, 1,0 ],      ! This flag implies whether or no
      GST_VALUE      = [ 5,0,32,0 ],      ! the record is an entry definition.
                                          ! The value of the global.
                                          ! (This won't work for PSECTs)
      ! The following 2 pairs are mutually
      ! exclusive. The first one is for GST
      ! records of type GST_GLOBAL_DEFN, the
      ! second is for GST_ENTRY_DEFN and GST_PROC_DEFN.

      GST_G_NAME_CS   = [ 9,0, 8,0 ],      ! The symbol name is a counted string.
                                          ! A dotted reference to this field
                                          ! picks up the count, an undotted
                                          ! one addresses the counted string.
      GST_G_NAME_ADDR = [10,0, 8,0 ],      ! The name string itself. An undotted
                                          ! reference is the address of the name,
                                          ! a dotted one is the 1st character.

      GST_E_NAME_CS   = [11,0, 8,0 ],      ! The entry name is a counted string.
                                          ! A dotted reference to this field
                                          ! picks up the count, an undotted
                                          ! one addresses the counted string.
      GST_E_NAME_ADDR = [12,0, 8,0 ],      ! The name string itself. An undotted
                                          ! reference is the address of the name,
                                          ! a dotted one is the 1st character.
      GST_P_MAX_ARG   = [1,0, 8,0 ],      ! Maximum number of formal arguments.
      GST_P_REM_CNT   = [1,0, 8,0 ],      ! Remaining byte count in arg. desc.
TES;

```

```

!+
! You declare an occurrence or REF of a GST datum via:
!-

```

```

LITERAL  GST_RECORD_SIZE = 27;    ! Each GST record is at most 27 bytes long.

MACRO    GST_RECORD = BLOCK[ GST_RECORD_SIZE, BYTE] FIELD( GST_FIELD_SET ) %;

```

```

!+
! The GST record types are defined as:
!-

```

```

LITERAL  ! GST types:

```

GST_LOWEST	= 1,	! We don't support global PSECTs now.
GST_PSECT_DEFN	= 0,	! P-SECT record.
GST_GLOBAL_DEFN	= 1,	! A global symbol definition record.
GST_ENTRY_DEFN	= 2,	! An entry point definition.
GST_PROC_DEFN	= 3,	! A procedure with formal arg. desc.
GST_HIGHEST	= 3;	! Highest one we support.

```

++
BLISS uses 'non-standard' DST records to encode
most of its local symbol information. These records
are like most DST records except that the TYPE
information is variable-sized.
--

```

```

FIELD
SET  BLZ_FIELD_SET =
      BLZ_SIZE      = [ 0,0, 8,0 ],      ! First byte is record size in bytes.
      ! The next byte contains DSC$K_DTYPE_Z, or we
      ! wouldn't be applying this structure to a given
      ! DST record.
      BLZ_TYP_SIZ   = [ 2,0, 8,0 ],      ! Type info takes up this
      BLZ_TYPE      = [ 3,0, 8,0 ],      ! many bytes.
      ! Which type of type Zero
      ! this corresponds to.
      BLZ_ACCESS    = [ 4,0, 8,0 ],      ! Access field.
      BLZ_STRUCT    = [ 5,0, 8,0 ],      ! Type of STRUCTURE reference.
      ! **** The following only work when BLZ_TYP_SIZ is 3.
      BLZ_VALUE     = [ 6,0,32,0 ],      ! DST VALUE field.
      BLZ_NAME_CS   = [ 10,0, 8,0 ],     ! The symbol name is a counted string.
      ! A dotted reference to this field
      ! picks up the count, an undotted
      ! one addresses the counted string.
      BLZ_NAME_ADDR = [ 11,0, 8,0 ]      ! The name string itself. An undotted
      ! reference is the address of the name,
      ! a dotted one is the 1st character.
TES;

```

```

++
You declare a REF to a BLZ_DST datum via:
--

```

```

LITERAL  BLZ_REC_SIZ      = 38;      ! Each DST record is at most 38 bytes long.

```

```

MACRO    BLZ_RECORD = BLOCK[ BLZ_REC_SIZ, BYTE] FIELD( BLZ_FIELD_SET ) %;

```

```

++
The type zero sub types,
as defined in CP0021.MEM
must be within the following
range.
--

```

```

LITERAL
      ! Type Zero Sub-Types:

```

BLZ_LOWEST = 1, ! Lowest variable type we support.
BLISS_Z_FORMAL = 1, ! Description of a ROUTINE formal.
BLISS_Z_SYMBOL = 2, ! A BLISS LOCAL symbol.
BLZ_HIGHEST = 2; ! Highest variable type we support.

PATRTS.REQ;1

16-SEP-1984 16:52:48.^{N 5}₉₃ Page 29

++
End of PATRTS.REQ
--

0299

DIGITAL EQUIPMENT CORPORATION
CONFIDENTIAL AND PROPRIETARY